<講演2>もつれたソフトウェアを解きほぐす数学

AUTHOR(S):

長谷川, 真人

CITATION:

長谷川, 真人. <講演2>もつれたソフトウェアを解きほぐす数学. 京都大学附置研究所・センターシンポジウム : 京都からの提言 -21世紀の日本を考える (第6回)-「混沌の時代に光を探る」 2012, 6: 13-22

ISSUE DATE:

2012-03-01

URL:

http://hdl.handle.net/2433/179448

もつれたソフトウェアを解きほぐす数学

京都大学数理解析研究所 教授 長谷川 真人

1. はじめに：ソフトウェアと数学

数理解析研究所の長谷川でございます。

今日は、ソフトウェアについてお話しします。ソフトウェアというと、音楽、映画、映像などのソフトのことかと思われる方もいらっしゃるかも知れませんが、今回はそうではなく、コンピュータに仕事をさせるための、**コンピュータソフトウェア（コンピュータプログラム）**についてお話ししたいと思います。

コンピュータソフトウェアと聞いて、「私はパソコンなんか嫌いです、触りません、だから、私には関係ないので今から居眠りします」という方もおられるかも知れません。でも、ソフトウェアというのは、皆さんのそばにあります。脇役かもしれないけれども、皆さんの携帯電話、家電

```
sum: addi $sp, $sp, -8    L1: addi $a0, $a0, -1
     sw   $ra, 4($sp)         jal  sum
     sw   $a0, 0($sp)         lw   $a0, 0($sp)
     slti $t0, $a0, 1         lw   $ra, 4($sp)
     beq  $t0, $0, L1         addi $sp, $sp, 8
     addi $v0, $0, 0          addi $v0, $a0, $v0
     addi $sp, $sp, 8         jr   $ra
     jr   $ra
```

アセンブリ言語で書かれたプログラム

製品、自動車、今や至る所に組み込まれております。ですから、コンピュータソフトウェアと無関係の生活というのは、不可能ではないにしても、現在の日本では考えにくいかと思います。

さて、そのソフトウェアというのは実際にはどんなものかと言いますと、たとえばこんなのが簡単な例です。謎の呪文がズラズラと並んでいますね。これは、アセンブリと呼ばれる、コンピュータへの命令を羅列したものです。ロードせよ、ストアせよ、足し算せよと呪文が並んでいます。これではあんまりなので、もうちょっとスマートに、人間が読み書きしやすいプログラミング言語で同様の仕事をするプログラムを書くと、以下のようになります。

```
let rec sum (x) =
  if x <= 0 then 0 else x + sum (x - 1)
```

関数型プログラミング言語で書かれたプログラム

実際に利用されているソフトウェアの多くは、現在ものすごく大きなものになっています。数万行から数百万行というのもあります。多くの人の手間ひまがかかっております。さらに、作るの

も大変ですけれども、いろいろ不具合が生じます。その不具合を直すことはもっと大変な話になってしまいます。しかも困ったことに、ソフトウェアというのは、とても大事な場所に使われていて、ほんのちょっとバグ（不具合）があるだけでいろいろな問題が起きます。ときどき新聞などで報道されますが、証券取引所のシステムであるとか、銀行のシステムであるとか、そういうものが本当にごく僅かなソフトウェアの不具合で止まってしまい社会大混乱が起きる。あるいは車の事故が起きた時に、犯人は車に搭載された機器を制御するソフトウェアではないかと、ソフトウェアに罪があったかどうかを調べるわけです。そうなってくると、ソフトウェア産業の方にせよ、我々のような研究者にせよ、頑張って正しいソフトウェアをちゃんと作るテクノロジーを開発しなくてはいけない。しかし現に不具合があるものは仕方ないので、そうしたら今度はソフトウェアの不具合をなんとか上手に見つけ出して上手に直す方法を考えなくてはいけない。けれど、それはとても難しいことです。

後で少し詳しくお話ししますが、ソフトウェアに関して我々が直面している問題というのは、だいたい、そう簡単な答えはない。機械的・自動的に解決できるようなものというのは、むしろ非常に少ないです。なんとか解決する方法が見つかったとしても、莫大な時間やスペースなど、大変な手間暇が必要になることもしばしばです。そうすると、計算量の問題がバカにならないのです。巨大なソフトウェアの場合、力ずくでそういう方法を使って解くというのは、現実的ではありません。

実は、皆さんが使っておられるソフトウェアの中で、全く不具合がないといわれているものは、まずありません。ソフトウェアというのは、大変苦労して作っておられて、出荷される前にもものすごく手間をかけてテストをしておられます。ソフトウェアテストの企業の方とお話をすると、非常に頭が下がる思いなのですが、しかしその会社の人たちが「我々のテストは完全ではありません」と認めておられます。バグが出そうな場合をテストでつぶしていくのですが、完全にというのは到底無理ですと言われます。そう考えますと、皆さんの携帯電話や自動車がちゃんと動いているのは奇跡のようなことかも知れません。

前置きが長くなりましたが、私は数理解析研究所という数学の研究所におります。そこで何をしているかというと、数学を武器にしてソフトウェアの問題に立ち向かう方法を考えようということやっております。ソフトウェアについてはともかく、数学は苦手、数学は無味乾燥でつまらない、数学ってもう終わっている話じゃないのと、よく言われるのですけれども、私に言わせればそれは全くの誤解であり、数学は我々の強い味方であって人類のこれまでの英知の結晶であると言って良いものだと思います。しかも終わっているどころではなくて、現在も目覚ましく発展しつつあります。今日のお話でそういった誤解を少しでも解けたらなと思っております。今回時間も限られていますので、２つのことにスポットをあててお話しようと思っています。

一つは、数学の特徴である「**正確さ**」。これはみなさん何となくそうかなと思われるかもしれませんけれど、数学の言葉はほんとうによくできていて、正確にソフトウェアの問題を定式化でき

るのです。これはとても大事なことで、たぶん数学以外では考えにくいです。

もう一つ、こちらはもっと力を入れて説明したいことですが、数学の得意なこととして**「抽象化」**があります。抽象化というと、具体性に欠けるとか、ネガティブなイメージを持たれますけれども、数学における抽象化というのは、問題のつまらない表面をバッサリ削り落として、本質に肉迫するために行う、正しい抽象化です。それは、ソフトウェアのややこしい複雑で巨大な問題に対しても有効です。

少しだけ研究所の宣伝をしておきますと、数理解析研究所は日本を代表する数学の研究所といって良いかと思います。純粋数学から応用数学までいろいろな分野で世界をリードする方をたくさん擁してきましたし、今も擁しております。私自身はそういう方々の足下には全く及びませんけども、今日お話しするようなソフトウェアの研究も、実は数理解析研究所の偉大な先達の影響を受けつつやっていることであります。

2. ソフトウェアの基礎としての数学

先程お見せしましたが、ソフトウェアというのはただの記号の列です。それを本当にコンピュータで役に立たせるには、いろいろな取り決めをしてやる必要があります。

ソフトウェアを書くためには**プログラミング言語**という人工的な言葉を使います。これはれっきとした言語です。言語ですから、書き方、文法というものがあって、文法通り書いた文章がどんな内容を持っているか、どうやって動くか、そういうことを決める仕組みがどんなプログラミング言語にも備わっています。ありがたいことに、プログラミング言語では、文法とか動かし方は完全に数学的に定式化が可能です。実際にやっているかどうかは別ですけれども、原理的には、こういうのは正しいプログラム・正しい文章であって、こういうのが実行した結果・内容であるということが、きちんと数学の言葉で定式化できます。この辺が英語や日本語のような自然言語とはだいぶ違うところです。例えば私がよく使っているプログラミング言語では、文法によれば、こういう文字列を書くとそれは正しいプログラムだ、ということが数学の言葉で定理として証明できます。言語の動かし方のルールを調べれば、このプログラムを実行すると、かくかくしかじかの足し算を実行するなどということが証明できます。プログラムを走らせなくても数学の定理として証明ができます。

ですから、ソフトウェア、プログラミング言語は、数学的にきちんと定式化することができます。原理的には必ずできます。そういうと、ソフトウェアの問題はすべて数学で解決できると思われるかも知れません。しかし、残念ながらそうではありません。定式化できるからといって、解決できる、ということではないのです。方程式というものがありますけれども、問題を定式化するというのは方程式を与えることに大体相当し、問題が解決できるということは方程式を解くことに相当します。方程式が解けるかどうかというのは、中学校などで習う方程式は解けますけれども、一般に数学の方程式は解き方がわかっていないものがゴロゴロあるわけです。だから解ける

かどうかは全く別の問題です。勝利の方程式というようなことを仰る方がおられますが、解けなかったら何にもならない。

解ける解けないに関して言いますと、ソフトウェアに関する問題というのは、大まかに分けると、解ける問題と解けない問題に分かれる。……当たり前ですね。解けるというのは、こういった手順をこなしてゆけば解けますと説明ができる、ということです。別の言い方をすれば、その問題を解くためのプログラムを書くことができるということです。一方、解けない、というのは、そんな上手いやり方が全然ない、ということです。実は、かくかくしかじかという手順でやれば必ず解けますよという教科書に書けるような、あるいはプログラムに書けるようなやり方が全く無い、解けない問題があります。困ったことに、ソフトウェアに関するほとんどの多くの問題は、どんなに頑張っても解けません。上手い手順が原理的に存在しない。これはずいぶん昔から、コンピュータサイエンスの初期から知られていることです。その最初の例が１９３０年代に指摘されています。ソフトウェアに、データを入力して、実行してみたとき、その計算が止まるか、暴走してしまうかということを判定しようとすると、これは解けない問題になってしまいます。実際にソフトウェアに入力を与えてじっと結果を待っていればいいじゃないかと思われるかもしれませんが、結果が出るのが１ 分後かもしれないし１ 年後かもしれない。１００ 年後かもしれない。下手をすると１０ 億年後かもしれない。そういうことを考えると、実際に走らせるというのは全く答えになっていないのです。

仮に、ソフトウェアが暴走するか、止まるか判定できる方法があるとしましょう。つまり、判定するプログラムが書けるとしましょう。そうするとそれを使って、ソフトウェアにソフトウェアを…プログラムは文字列ですから入力として与えることができます。ソフトウェアに自分自身を入力させたら止まるかどうかを判定するプログラムが書ける筈です。そういうソフトウェアがあると、さらにちょっと細工して、自分自身を入力して、暴走する場合には止まる。止まるという場合には暴走するというひねくれたソフトウェアを書くことができる。それで、このひねくれたソフトウェアにひねくれたソフトウェア自身を入力して与えてやった時に、どういう結果が出るか。ご存知の方もおられるかもしれませんが、「私は嘘つきだ」という人は嘘つきか正直かよくわからない。どっちにしても矛盾が出るのですが、同じ様に、このソフトウェアも止まるとすると止まらない。止まらないとすると止まるというパラドックスが生じます。つまり、ソフトウェアが暴走するかを判定するソフトウェアは存在しないのです。

こんなネガティブなことを言ってしまうと、数学ではだめじゃないのとか、数学ではソフトウェアの問題はやっぱり解けないのですかねと言われますが、すべてそうというわけではない。解けるものもあります。それから、解けないとわかっている問題であっても、部分的には解ける、全ての場合は無理でもこういう条件を課せれば解ける、というのはたくさんあります。実際、ソフトウェアに関する問題では、応用上はこういう仮定を置けばそれで十分役に立つ、というようなことはよくあります。そうするとコンピュータサイエンスの腕の見せ所としては、そのような上

手い解決法を与える、あるいは上手い解決法があるような枠組みを考えるという、そういうところになってきます。

3. ソフトウェアを解きほぐす数学

ここからは、私が実際に取り組んでいる「ソフトウェアを解きほぐす数学」という話題についてお話したいと思います。プログラムが普通の文章と違うことのひとつに、頭から最後まで順番に読んでいくというものではなくて、プログラムに書かれていることを解釈して、あちこち計算の流れが飛び回る、ということがあります。そういうのを制御構造と呼びます。制御構造はプログラミングでは大事な機能ですが、濫用すると、あっという間にわけのわからないプログラムになります。業界ではスパゲティプログラムと言って、こんがらがったスパゲティに例えることがあります。

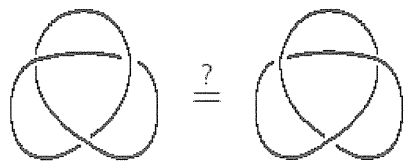
例えば、冒頭の関数型プログラム、これは「再帰」とよばれる制御構造ですが、プログラムが自分自身を使って仕事をするように書かれています。自分自身を呼び出して仕事をする。これは変に思われるかもしれませんが、実際にプログラム書くとすごく当たり前のことなんです。当たり前のことなのだけれど使いこなすのは大変で、再帰プログラムを実行したら何が起きるかを正確に予測するのはとても難しい。

また、パソコンを使っておられる方でしたら、今まで何かのソフトウェアを使っていて、ある時ソフトを新しいバージョンに更新したら動かなくなった、というようなことを経験されているかと思います。こんなことは世界中どこでも起きているのですが、困った事に、そういうのは一般的には「解けない問題」になのです。どういうことかというと、２つのソフトウェアが同じ働きをするのかどうか、一方を一方で置き換えて不具合が起きないかどうかを判定するうまい方法というのが一般にないので、できる事として部分的な解決を探す訳です。

私の研究は**プログラミング言語の意味論**というもので、プログラムの内容を読み解く仕事をやっています。プログラムに、ある種の抽象化を行って、いらない情報を削り落としていって、より扱いやすい情報の形にしてやって良い結果を得る、という理論です。システマティックに読み解くということです。目標としては問題の本質を浮き彫りにするところに重きがあります。これは、数学ではわりと普通の考え方で、数学者というのはいつもそういうことをやっていて、偉いなと思うのですけれども、数学では**「不変量」**といいます。

例えば幾何学という分野があって、素人から見ると、幾何学というのはややこしい図形を一生懸命ながめたり、測ったりする分野かと思ってしまうのですが、それは違います。そうではなくて、図形にあれやこれやいろんな変換、操作をしてやり、どんな操作をしても変わらない性質に着目するのが幾何学なのです。これはクラインという数学者が言ったことです。クラインの壷という言葉をご存知の方もあるかもしれませんが、そのクラインさんが、ずっと昔、１００年くらい前に看破した、現代の数学の基本的な考え方の一つです。

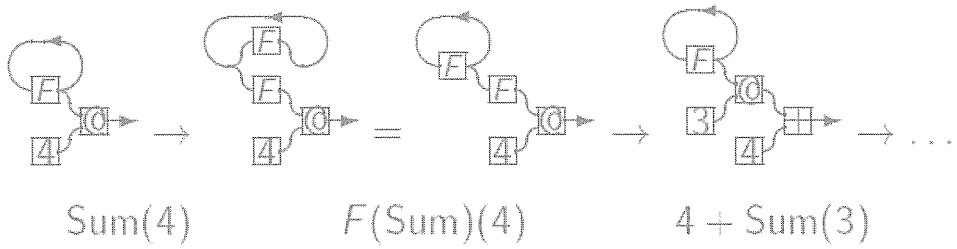
わかりやすい例として、ゴム面の幾何学とかトポロジーとか呼ばれる分野の話題に、**「結び目の理論」**というのがあります。結び目の理論とはどういうのかと言いますと、ひもの両端をつないで閉じちゃうんですけれども、ややこしくこんがらがったりするのです。そのようにして作った結び目が２つあるとき、これらは適当にぐにゃぐにゃとやっていったら同じものになるかどうか、ということを考えます（下図）。



これは難しくて面白い問題ですが、我々素人は一生懸命ぐにゃぐにゃとひもをこねくりまわして、ああでもないこうでもないとやってしまうのですが、トポロジーの専門家は、それをする前に、ぐにゃぐにゃ変形する操作に関して変わらない性質に着目します。それを**結び目の不変量**といいます。

我々のソフトウェアの研究でも、全く同じでありまして、ふたつのプログラムが同じ働きをするかどうかを知りたい時に、プログラム上の適当な操作に関して変わらない数学的な量というのを見つけてくればいいのです。これがさっき言ったプログラミング言語の意味論ということ。内容をシステマティックに読み解くということです。変わらないものを見つけ出すのが大切なのです。

実は、結び目の話と同じような事を、プログラミング言語についても考えることができます。さきほどお見せした関数型プログラミングの例。中身はあまり説明しませんけれどもこれも実は、自分自身を呼び出す再帰プログラム。自己言及のあるプログラムでして、自己言及のあるところを輪っかにして、Ｆというのは自分自身で、このＦに向かってぐるっと閉じた輪っかを描いて、それで自分自身を使いますよということを表して以下のような絵が描けます。



細かい事は説明しませんが、プログラムの実行というのは、こういう絵を適当な規則に沿って書き換えていく、そういう操作に思えなくもない。そうするとさっきの結び目の問題、２つの結び目が一緒かどうかという話と、この２つのプログラムが同じ振る舞いをするかどうかという話とは似ていますよね。

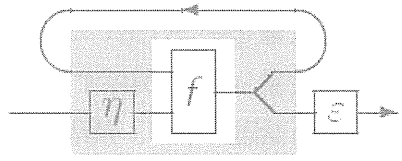
プログラムを変形させたとき、変形に関して不変な量というのは、きっとそのプログラムに関して本質的な情報を持っている筈です。それをどうやって求めるのかが問題でして、これは私なんかが始めるずっと前から、いろんなアプローチが知られていました。だいたい４０年くらい前からとられている領域理論と呼ばれるアプローチでは、プログラムの情報の量の大小を比較して、その大小の関係の情報からうまいこと不変量を見つけ出したりしています。最近になって、このアプローチが、今お話しした「結び目の不変量」と深く関係するという事がわかってまいりました。

結び目の不変量というのは大昔からある話題ですが、飛躍的に発展したのは１９８０年代からです。何が起きたかというと、数理物理学のアイデアを取り入れると、うまいこと良い不変量がたくさん出てくるということがわかった。量子論に関連して発見された不変量なので、**量子不変量**と言ったりします。ちなみに、その量子不変量についてとても大事な「**量子群**」という概念があるのですが、これは、１９８０年代、数理解析研究所におられた神保道夫先生が、ロシアの研究者と独立に発見されたものです。その量子群から、結び目の不変量がいくらでもみつかるということがわかって、大変さかんに研究されてきました。今量子不変量についてくわしくお話しすることはできませんが、その非常に単純化されたケースとして、行列の計算を少しだけやってみましょう。

高校で行列のかけ算を勉強されたと思いますが、行列はかける順番によって答えが変わります。

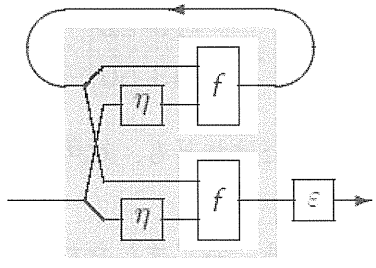
$A=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}$ と $B=\begin{pmatrix}5 & 6\\ 7 & 8\end{pmatrix}$ をこのまま並べてかけ算すると $AB=\begin{pmatrix}19 & 22\\ 43 & 50\end{pmatrix}$ 。Ｂ と Ａ という順番でかけ算をすると $BA=\begin{pmatrix}23 & 34\\ 31 & 46\end{pmatrix}$ 。違いますね。違うんだけれども、対角線だけみてやって足し算をすると、どちらも６９ になります。ああ、たまたま一致しましたね、と思われるかもしれません。しかし、実は、どんな行列を持ってきてかけ算の順番を入れ替えても、対角線のところの足し算の結果は必ず一致します。これは大学１年程度の数学で証明できます。この結果は、結び目の不変量に関係しています。かけ算の操作は、A というものと B というものをつなげる操作に相当していて、対角線のところの足し算をとるというのは、つなげてきたものをぐるっと輪っかにする操作に対応します。ですので、AB の対角線の和と BA の対角線の和が等しいというのは、A と B を順番につなげてぐるっと輪っかをつくったものと、B と A をつなげて輪にしたものとが、ぐにゃぐにゃぐにゃっと線のところを引っ張ったり縮めたりすれば同じものになるということに相当しています。実際、このような行列の計算は、結び目の不変量を単純化した例になっているのです。

実は、再帰プログラムについても、同じ発想から意味論＝不変量を与えることができます。どういうことかというと、行列の対角線の和をとる操作をもう少し抽象化した数学を使えば、プログラムに対しても結び目と同じような考え方ができます。絵に描くとこんな感じです（1）。

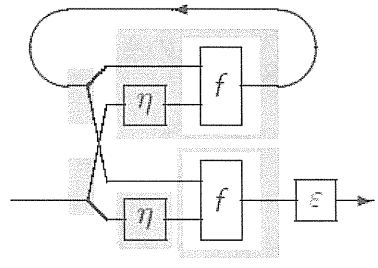


（1）

灰色に囲まれた、いくつかの部品をつないだものを、プログラムだと思うことができます。これはちゃんと意味がある絵でして、数学的な規則に沿って読み解いていくと、これは再帰プログラムをあらわしていることがわかります。このことを、これから数学的紙芝居にしてお見せします。まず、枝分かれをしているところがあります。f から右に出て枝分かれしているところ。そこでガサッとコピーします（2）。コピーすると絵が大きくなりますが、今度はその灰色のところを切り刻みます（3）。

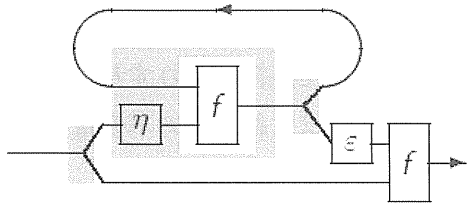


（2）

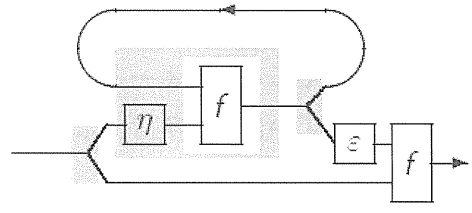


（3）

だいぶバラバラになりました。つぎに、絵の下の方を眺めてやると、規則に従って変形ができて、スッキリします（4）。スッキリしたところで絵を結び目みたいにぐにゃぐにゃと変形をすると、こんなふうになります（5）。

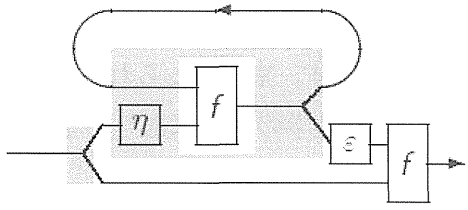


（4）



（5）

最後に、灰色の部分をつなげてわかりやすくします（6）。



(6)

じっと最初の絵（1）と最後の絵（6）を見比べると、f が、ひとつ余計に右の方に出てきていることがわかります。詳しいことは述べませんが、これは、自己言及的なプログラムの動きを、きちんと数学的に説明したものになっています。コンピュータソフトウェアの背後に、結び目と同じ構造が潜んでいたというのは、ちょっと意外なことかも知れません。このような、隠れていた本質を見出すことは、プログラミング言語の意味論の醍醐味であると思います。

4. まとめ

私たちの取り組んでいるソフトウェアと数学の関わりの、ごく一部についてご説明いたしました。ソフトウェアの問題に立ち向かうために、数学の正確な定式化の力と、柔軟な抽象化を図る事がきちんと役に立っているということを感じていただけると大変ありがたく思います。また、数学は、ソフトウェアに限らず、社会の大事な脇役となって威力を発揮している私たちの頼もしい味方であるということを、是非知っておいていただけたらと願っております。

もう少しだけ本音をお話ししますと、実際にはそんなきれいごとですむ訳ではなくて、毎日苦労しております。数学というのは美しく深いところで良い結果が出るようになっていて、ですから数学者は美しさをとても大事にしています。その美しさと言うのはソフトウェアの本質ともつながると私は信じていますけれども、そうはいっても、美しい数学と、ソフトウェアの現場で直面する生々しい問題とをうまく結びつけるのは、私にとっては本当に難しいことです。だからこそやりがいがあるわけですが。

私は、数学を使って、ソフトウェアの理論的な側面の研究に力を入れておりますが、数学、あるいは科学においては、永遠不変の真理を究めることが一番の関心事であるかと思います。変わらぬ真理を求めることは何よりも大事なこと。一方で、ソフトウェアの世界というのは驚くほど移り変わりが速い。数年で陳腐化してしまう。パソコンだって、20年前のものは、もう化石といってもいいですよね。４０年前のコンピュータで、現在動いているのはまずないであろうと思います。原発とは全然違うわけです。ソフトウェアというのは、善くも悪くもどんどん変わっていく世界です。そういう変わっていく世界というのはエキサイティングで面白いけれど、それと、じっくり腰を据えて真理の追究というのとは、大分違うわけですね。１００年後に通じるコンピュータサイエンスをつくろうという心意気と、一方で来週の納期どうしましょうという話とは、なか

なか両立は難しい。できている方もおられると思うのですが、私はまだまだでして、力及ばず歯がゆい思いをしております。

私なりに少しまとめさせていただきます。京大におられた有名な物理学者の佐藤文隆先生が本に書かれていることを引用します：「物理学の学習はありのままに世界をみない手法を身につけるための修行である。」

物理学というのは、林檎が木から落ちるところを見る学問ではないわけですね。その裏に隠れている真の宇宙の姿、本当の法則性を見いだす。そういう分野だろうと思いますけれど、私も佐藤先生の真似をして、「ソフトウェアの理論の研究は、ありのままにソフトウェアを見ない手法を身につけるための修行である。」と言いたいです。現実には、目の前のソフトウェアをしっかり見ないと明日までにバグが取れないということはありますけども、それはそれとして、１００年後まで通じるソフトウェアの学問をつくるのであれば、ソフトウェアの理論の研究というのは、ちょっと目を閉じ、ソフトウェアをありのままでは見ないで、表面にとらわれないで、本当の中身、本質に目を凝らす。そういうことが大切なのではないかと、自分を励ましてやっております。

駆け足になってしまい、申し訳ありませんでしたが、以上でおしまいにしたいと思います。